Trie is an extremely efficient data structure for the purpose of adding words to it and searching if a certain word exists in the Trie or not and many similar situations. Tries can also be used to store binary representation of numbers and one of the common algorith used in conjunction with storing bits of numbers is find max XOR of two integers.

In a normal word/string based Trie we perform primarily three operations.
- Insert a new word
- Search if a word exists in Trie.
- Search if any word in Trie starts with a specific prefix.

All other problems related to words & Tries usually make use of some variation of above methods.
A typical Node class for Trie looks something like

```
Node {
  Node[] links;
  boolean flag;

  Node() { this.links = new Node[26]; }
}
```

Here, the links array will basically store the reference to a child Trie for a certain character.

> Note:- In above Node class we are assuming we will only have 26 english alphabets in our words/string. If we have more or less possible characters we can adjust the size of links accordingly

There are some common operations which will be performed on Node class for which we can use some simple helper methods insid node class.

-> contains key method - This method will take a character as input & check if we have reference Node at that position. Assume a is 0, b is 1 . . . . z is 25.

```
return links[character - 'a'] != null;
```

-> get(char el){ return links[el - 'a']; }

-> put(char el) { links[el - 'a'] = new Node(); }

-> isEnd() { return flag; }

-> setEnd() { this.flag = true; }

Once, we have our Node ready we can easily implement above discussed methods.
Create root node of trie after
- Insert
  - Store root node in an node variable.
  - Loop through all the characters in the given string
  - If node does not contain the character we call put method & create new reference node.
    - We update node to node.get(el); basically advancing in our Tree (if we already had that node we wouldn't create new node).
  - At the end we know the node points to reference node at last character in our given string. For this node we can call the setEnd() method.

- contains(String word)
  For this we follow same steps as previous method. The only difference is while looping through all the characters if node doesn't contain any key we simply return false.
  Even at the end we can just return node.isEnd() because for word to exist this flag needs to be set as true.

- startsWith(String word)
  Exact same as contains. Only difference being if we have not found contains key as false for any character we can simply return true.

One note is that we don't always need flag in node. If we need the entire word we can also just store string(word) end node. The point is to modify it according to our needs.